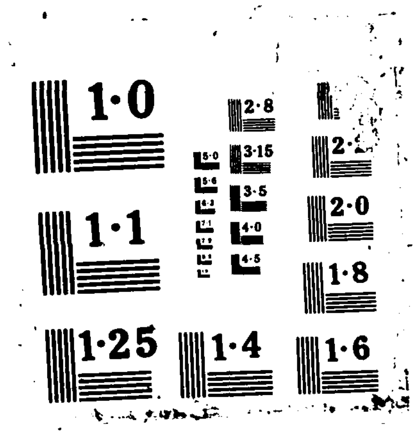
AD-A192 449 AUTOMATIC ERROR RECOVERY IN A FAST PARSER(U) COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE R W GRAY 1987 1/1

UNCLASSIFIED F/G 25/4 NL


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·8
3·15
3·5
4·0
4·5
5·0
5·6

Automatic Error Recovery in a Fast Parser

Robert W. Gray
Computer Science Dept 430
University of Colorado
Boulder, CO. 80309-0430
bob@boulder.COLORADO.EDU

ABSTRACT

Although parser generators have provided significant power for language recognition tasks, many of them are deficient in error recovery. Of the ones that do provide error recovery, many of these produce unacceptably slow parsers. I have designed and implemented a parser generator that produces fast, error recovering parsers. For any input, the error recovery technique guarantees that a syntactically correct parse tree will be delivered after parsing has completed. This improves robustness because the remaining compilation phases, such as semantic analysis, will not have to deal with infinitely many special cases of incorrect parse trees. The high speed of the parser is a result of making the code directly executable and paying careful attention to implementation details. Measurements show that the generated parser runs faster than any other parser examined, including hand-written recursive descent parsers. The cost of this fast parser with error recovery is a slight increase in space. Although this particular generator requires LL grammars, the ideas can be applied to generators taking LALR grammars. Furthermore, we give the transformations that allow one to transform many LALR grammars into equivalent LL grammars.

1. INTRODUCTION

Although parser generators produce code to recognize the structure of input, many such as YACC [Johnson1979] do not provide syntactic error recovery. The PGS [Dencker1985] tool generates an error recovering parser, but one that runs so slowly that it is a major bottleneck in the overall analysis [Gray1985]. This efficiency penalty is severe enough to dismiss PGS as a useful production quality tool, and compiler writers are forced to do without automatic error recovery. In this paper, I describe the design and implementation of DEER, a tool that produces fast parsers with error recovery. The high speed is achieved by producing directly executable code as opposed to the common practice of producing tables that are interpreted at runtime. The idea of direct execution is not new; recursive-descent parsers have always had significant speed advantages over table driven parsers. The significant contribution of DEER is that it generates from a grammar, a fast parser, with error recovery. Complete details of the ideas presented herein and the code which implements them are contained in *Generating Fast, Error Recovering Parsers* [Gray1987].

The most important part of a compiler in terms of user interface is error recovery. It is unacceptable to merely announce the first syntactic error of an input program and quit. All syntactic errors should be reported when a compiler is run; furthermore, semantic errors should also be reported even if syntactic errors are present. Clearly, a simple syntactic error such as missing punctuation should not preclude semantic error processing. The difficulty here is that semantic analysis must have a syntactically valid parse tree[1] in order to reliably proceed. Ad hoc error recovery cannot guarantee that a correct parse tree will be passed to semantic analysis — there are far too many special cases that need to be handled. This is why we consider a YACC generated parser to be deficient: when it detects an error a user written recovery routine is called. Often, users of YACC will not take the time to write an error recovery routine and in this case the parser will terminate at the first error. Implementing complete and reliable recovery is involved and tricky; therefore, it should be automated. We desire a formal recovery technique, one based on sound theory. The same technique that Röhrich [1980] has implemented for PGS is used for DEER. The recovery is good for most syntactic errors although there are cases where hand tuned recovery will do better. Still, because of the guarantee of correctness, plus the *no work required of the programmer* feature, automatic error recovery is the method of choice.

Section 2 describes how very fast parsing can be achieved through direct execution. Section 3 presents the recovery technique and how it has been incorporated it into the very fast parser. Performance measurements in Section 4 confirm the advantage of direct execution. Section 5 discusses grammar transformation.

2. DIRECTLY EXECUTABLE PARSER

This section addresses the problem of generating a very fast parser. We know that recursive descent parsers are fast. These parsers are based on LL(1) grammars, so it should be possible to *generate* a parser that runs as fast as a recursive descent parser. As it turns out, a DEER generated parser runs significantly faster.

The key insight for high speed parsing is that the parser should be directly executable. Recursive descent parsers are directly executable — there is a procedure to recognize each production of the grammar. Transitions of the automaton are often implemented with jumps, for example a PASCAL parser might contain

```
if ( lookahead == 'BEGIN' )
        begin_processing_code
else if ( lookahead == 'IF' )
        if_processing_code
...
```

On the other hand, most parser generators (including YACC, SYNPUT, PGS, Bison) produce tables that must be interpreted. The automaton of such a parser enters the next state by looking in a table, and performing the action.

$$\text{table} : \text{state} \times \text{symbol} \rightarrow \text{action}$$

On the average, this takes at least 3-4 times longer than direct execution. Therefore, to achieve very fast parsing, DEER produces executable code, not tables.

Figure 1 gives a fragment of a DEER generated parser. It is similar to a an assembly language listing of a recursive descent parser. The code for each case corresponds to code of a procedure in a recursive descent parser — one per non-terminal of the grammar. A call to these pseudo procedures, is implemented as the sequence *'PU( ); goto L;'* and a return is *goto pppop*. Notice that the case labels are compact. This is one of the implementation details that can yield an improvement in execution speed of up to 25%. *IF_NOT* is a carefully coded macro that generates a single machine instruction to test whether the lookahead symbol *la* is in the set referenced by the first argument. For example, D2 might be the set of terminal symbols that

[1] Often, there is no reason to form an explicit parse tree, instead a prefix linearization may be more appropriate (see *connection points* in [Waite1983]).

initiate statements: { *IF, BEGIN, WHILE, REPEAT, ...* }. When the parser cannot accept the lookahead in its current state, a call is made to parseErr, which carries out the error recovery. When it returns, normal parsing resumes.

```
goto L0;
pppop:              switch (--*DEPSp) {
case 0:             break;
          L0:       IF_NOT(D2,la) la=parseErr(la,L0);
                    goto L1;
          L1:       PU(1); goto L14;
case 1:
          L2:       goto pppop;
          L3:       if (la != int) goto L6;
          L4:       la = nexterm( );
          L5:       goto L13;
...       ...       ...
case 3:
          L25:      if (la != ]) la=parseErr(la,L25);
                    la = nexterm( );
          L26:      if (la != of) la=parseErr(la,L26);
                    la = nexterm( );
          L27:      IF_NOT(D2,la) la=parseErr(la,L27);
                    PU(4); goto L14;
case 4:
          L28:      goto pppop;
  }
```

Figure 1. Directly Executable Parser

At first glance, the directly executable parser appears to have a structure similar to an interpretive parser — a big switch statement. However, an interpretive parser requires a loop iteration for every state change: the directly executable parser makes most transitions with sequential execution and goto statements. A switch statement is required for only a fraction of the cases.

Figure 2 shows the process of building both a YACC and a DEER parser. Again, the major difference is that *y.tab.c* is mostly a file of tables that will be interpreted whereas *files* is mostly directly executable C code.

3. ERROR RECOVERY

Now that we have obtained high parsing speed, the problem is to incorporate error recovery without diminishing parsing speed. The error recovery should be automatically generated from the parsing grammar without any extra effort from the user of DEER. The quality of its syntactic error messages is an indication of the user friendliness of a compiler. In the best case, the user is immediately led to all syntactic errors of his program. In the worst case, if error recovery operates incorrectly, the compiler may crash, leaving the user helpless.

Section 3.1 gives a motivation for automatic error recovery and brief overview of existing techniques. Section 3.2 informally presents the ideas behind the DEER error recovery method.

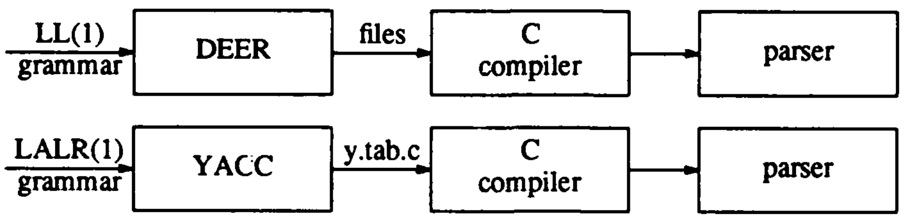


Figure 2. Steps to build YACC and DEER parsers

3.1. Overview

As a parser operates, it consumes input, token by token. The consumed input, also called *accepted* input, drives the parser into a particular state. Deterministic (non-backtracking) parsers never accept a token that cannot legally continue what has already been accepted. This is one of the principle merits of LL and LALR parsing techniques — they are guaranteed to detect errant tokens as soon as they are encountered.

The user should receive as much information as possible from each compilation attempt. It is unacceptable just to detect the first error and quit. The parser should repair errors and continue parsing. Finally, it should deliver a valid parse tree or connection sequence to the rest of the compiler. If the parser tries to recover but delivers a faulty parse tree, the remaining phases of the compiler could crash and leave the user helpless.

Gries [1976] gives an excellent annotated bibliography for error handling. Also, Horning [1976] presents an overview of various techniques of error handling. There are many strategies a parser can employ for error recovery: panic mode, phrase level, error productions, global, and automatic.

One of the simplest language independent recovery techniques is the *panic mode*. When an error is detected the input is skipped until one of a predefined set of "special" symbols such as *begin* or ";" is encountered. The parsing stack is popped until the special symbol can be accepted. Unfortunately, this method has many shortcomings. It frequently results in deleting large portions of the source text. In addition, semantic information depending on the erased part of the stack becomes inconsistent. Finally, the set of special symbols must be determined by hand.

On detection of an error, *phrase-level* recovery makes a backward move in the parse stack and a forward move in the remaining input. This isolates a phrase which is likely to contain the error. Then a weighted minimum distance correction is carried out at the phrase level.

Joy, Graham and Haley [Graham1982a] use *error productions* for their production Pascal compiler. First the compiler writer needs to predict the most likely kinds of errors expected. Then error productions must be written by hand. It is impossible to foresee all error conditions. In practice, the parser must be exercised to see how well the error recovery works. Further tuning is likely to be needed. There is another shortcoming of this method: the added error productions could make the grammar ambiguous.

Global error recovery attempts to find the smallest set of changes that will make a given program syntactically correct. It is impractical from the perspective of efficiency due to the exponential number of corrections that must be considered. The smallest set of changes if there are more than two errors is to enclose the error portion with comment brackets. This is usually not a desirable recovery.

Röhrich [1980] has implemented *automatic* construction of error handling parsers for LALR(1) grammars, and Fischer [1980] has done it for LL(1) grammars. The technique is based on a sound theoretic foundation. The resulting parsers are capable of correcting all syntax error by insertion and/or deletion of tokens to the right of the error location. Therefore, no backtracking is needed, and the output of the parser always corresponds to a syntactically valid program. This contributes significantly to the reliability and robustness of a compiler. The speed of parsing correct parts of a program is not affected by the presence of the error handling capability.

Röhnch's technique of automatic error recovery was chosen for incorporation in the directly executable parser because it automatically derives the error recovery directly from the grammar. Many other techniques require the manual specification of error recovery.

3.2. Automatic Error Recovery

In this section, I give a high level, intuitive presentation of automatic error recovery. To affect error recovery, the parser will delete zero or more tokens and generate zero or more tokens, then normal parsing will resume. The simple grammar of Figure 3 will be used for a few examples.

type	$\rightarrow$	simple
	|	**^id**
	|	**array** [simple] **of** type
simple	$\rightarrow$	**int**
	|	**char**
	|	**num .. num**

Figure 3. Sample grammar

In the following input, the errant token which we will call t is the second left bracket.

array [[int] of char

Our recovery technique will delete the second left bracket and resume normal parsing. For input,

array [int of char

the errant token "of" cannot be accepted in the current parse state. However, after a close bracket is inserted, then it can be accepted. For the input

array [[char

the second left bracket will be deleted, and the generated string of tokens "int] of" will be inserted.

A complete and formal description of the error recovery technique can be found in [Gray1987, Waite1983, Rohrich1980]. For the motivated reader, the rest of this section provides the essential details.

A parser for language L will *accept* input strings (i.e. programs) in L. Let T be the set of terminal symbols of the language L; then $T^* - L$ is the set of all erroneous programs. Let $\omega t \chi$ be an erroneous program, where ω is an initial string that is syntactically correct and has been accepted by the parser and symbol t cannot be accepted by the parser. The rest of the program is the string χ. We say that t is a *parser-defined* error.

If $\omega t\chi \in (T^* - L)$ is an erroneous program with parser-defined error t, then to effect recovery the parser must alter either ω or $t\chi$ such that $\omega' t\chi \in L$ or $\omega t'\chi' \in L$. Alteration of ω is undesirable since it may involve undoing the effects of previous actions. It is too expensive to retain information in case backtracking is needed. Thus, we consider the alteration of only t and χ.

In more detail, the error recovery works as follows: A fixed terminal symbol $f(q)$ is associated with each state q of the parser. When an error is detected, the parse stack is copied. Then a "continuation parse" is carried out using the copied stack and $f(q_i)$ as input at each state q_i. In addition the set of allowable terminals (*Director set*) of each state q_i is added to the *anchor* set which is the set of all terminals that could be accepted during this continuation parse. The function f is chosen such that this process would terminate the parse rapidly, driving the parser through states $q_1, \ldots, q_n$. Next zero or more of the actual input symbols are discarded until an input symbol t'' is found which is in the anchor set. The state for which t'' is acceptable is q_i. Then, the error is corrected by inserting $f(q_1) \ldots f(q_{i-1})$ into the input stream to the left of t'' while adjusting the original stack. Finally, normal parsing is resumed.

4. PERFORMANCE

The time and space requirements of a parser can vary widely. Seemingly small details can make huge differences. In this chapter, I bring out some of the performance issues and then compare DEER and YACC generated parsers recognizing PASCAL.

4.1. Performance Details

Conventional wisdom for software tuning is to build a system, measure it, and then work on the areas which can yield the largest payoffs. I have used this approach. The design and implementation of DEER has been heavily biased toward fast parsing. Often, clever data structures reduce both time and space requirements; however, when there has been a conflict, I have chosen to trade off some extra space for higher speed.

Figure 4 gives the static frequency distribution of directly executable code to parse PASCAL. There are total of 730 such instructions.

174	la = nexterm( );
108	IF_NOT( ) la=parErr( );
92	if (!=) goto L;
88	if (!=) la=parErr( );
85	goto L;
82	PU( ); goto L
36	goto pppop;
...	...

Figure 4. Frequency distribution of code

The macro IF_NOT, which tests set membership, occurs very frequently and should be fast and compact. One of the original implementations expanded this macro into about 5 machine instructions. The current version, expands the test into one instruction. This saves about 1000 bytes for PASCAL (2 bytes per instruction * 5 instructions per test * 108 tests). There are similar speed advantages to the one instruction implementation.

The director set representation is crucial to data space efficiency. There are roughly 64 director sets and 64 symbols for PASCAL. The naive implementation would require about 4096 bytes. Bit packing reduces this to 512 bytes.

We keep the lookahead token in a register since it is accessed so often. A register is also used for the base address of the data structure that the IF_NOT macro references.

The static frequency distribution of code gives us no clue as to how often these statements are executed. It turns out that director set membership is heavily used. For the input program described in the next section, the IF_NOT macro is used 37,267 times and the PU macro is used 19,590 times.

There are a number of other examples where careful tuning can yield substantial time and efficiency payoffs. These include choosing registers for heavily used objects, such as the lookahead symbol. The stack pointer is also placed in a register.

4.2. Comparison

This section compares the time and space requirements of the parsers. All measurements were carried out on a SUN 3/75 running SUN UNIX 3.2. The parsers, which are written in C, were compiled with the optimize flag (-O). The call graph execution profiler *gprof* [Graham1982b] and *time* provided the speed measurements. The *size* command provided the space information for *text*, (the executable code), *data*, (the initialized data), and *bss* (the uninitialized data, zero fill on demand). The input used was the distributed SYNPUT pascal program. The file which is 105,813 bytes long, consists of 16,170 tokens. The token distribution of this input program is given in Figure 5.

5638	Identifiers
1869	;
965	:=
834	,
740	) (
400-523	Int : ^ .
200-368	END BEGIN String THEN IF] [=
70-182	+ NIL DO <> ELSE VAR
40-61	- PROCEDURE WITH WHILE
...	...

Figure 5. Lexical classification of input

The DEER parser has automatic error recovery; the YACC parser has no error recovery. Figure 6 compares time and space requirements of the parsers for the input. (The link editor rounds up sizes to the next 2k byte page boundary).

Parser	Total time	Parse time	Space			
			text	data	bss	total
DEER	2.0	0.32	40960	24576	6148	71684
YACC	3.2	1.24	24576	24576	5884	55036

Figure 6. Time and space requirements

The gprof tool extracted the parsing time from the overall time. DEER parses about four times faster than YACC. This speed advantage plus error recovery costs about 25% more space (71K :: 55K).

5. GRAMMARS

The DEER parser generator requires an LL(1) grammar, but many existing grammars are LALR(1). This section will make the following points:

(1) There is a need for a generator that produces fast, error recovering parsers using LALR(1) grammars. (We are currently working on this).

(2) When designing new languages, there are good reasons to use only LL(1) parsable constructs.

(3) Most LALR(1) grammars can be transformed into equivalent LL(1) grammars.

There are plenty of existing LALR(1) grammars and it does not make sense to transform these into LL(1) grammars just to have automatic error recovery. However, it would be desirable to have a generator that could use these existing grammars and produce fast, error recovering parsers. It could be based on an existing generator such as PGS or YACC. Corbett's [1985] Bison, which is similar to YACC, holds promise as a starting base because of its efficiency and clarity; furthermore, it is in the public domain. The major question is whether it would be easier to add error recovery to YACC or Bison, or make PGS faster. Pennello's [1986] work on making LALR parsing very fast should be considered before such a project is undertaken.

The trend has been for designers to specify more complex and more powerful languages. It is important to avoid using constructs that are difficult for humans to work with. Hoare [1981] recommends single-pass top-down recursive descent both as a implementation method and as design principle for a programming language. He says we want programs to be read by *people* and people prefer to read things once in a single pass. He concludes that there are two ways of constructing a software design: one way is make it so simple that there are *obviously* no deficiencies and the other way it to make it so complicated that there are no *obvious* deficiencies. The first is far more difficult. Wirth [1985] has similar views. He believes that the real cost of an overly complex language is hidden in the unseen efforts of the innumerable programmers trying desperately to understand them use them efficiently. For his most recent project, he choose the simple recursive descent top-down method which is easily comprehensible and unquestionably sufficiently powerful, if the syntax of the language is wisely chosen. Philip Machanick [1986] contends that the restrictions of LL parsers discourage the adoption of language constructs difficult for the human reader to comprehend. As a grammar becomes more and more powerful, its sentences become harder and harder to understand.

There are just a few constructs that are LALR, but not LL. *Left recursion*, which is not allowed in LL grammars, is often used in an LALR grammar to specify iteration, for example:

idlist → idlist, identifier | identifier

An equivalent and more transparent extended BNF grammar acceptable to DEER is:

idlist → identifier (, identifier)*

In LALR grammars, arithmetic expressions are typically specified using left recursion as shown with the following 3 lines.

```
E    →    E + T | T
T    →    T * F | F
F    →    ( E ) | id
```

The equivalent, but less transparent LL grammar is given in the next 5 lines.

```
E    →    T E'
E'   →    + T E' | ε
T    →    F T'
T'   →    * F T' | ε
F    →    ( E ) | id
```

In the following example, known as the *dangling else*, a parser needs to decide to which *if* the *else* belongs.

```
if expression
if expression
        statement
else statement
```

The grammar is ambiguous because two different parse trees can be built. With a more powerful class of grammar (such as LALR), the ambiguity can be removed by rewritting the grammar. However, the resulting grammar is unwieldy because it contains repetitions of the original simpler grammar. The preferred approach for both LL and LALR grammars is to use an ambiguous grammar, and some mechanism that allows the ambiguity to be resolved in a natural way.

Grammars that contain alternates which can derive a common prefix are not LL. This is because it is not possible to decide which alternate to choose (unless the lookahead is extended and the common prefix is of a fixed length). The grammar can be made LL by a technique called *left-factoring*. In a simple case, such as:

```
statement    →    identifier parameter_list | identifier := expression
```

left-factoring is straightforward:

```
statement    →    identifier stmt_follow
stmt_follow  →    parameter_list | := expression
```

6. CONCLUSIONS

It is possible to produce parsers that are both fast and that have error recovery. Based only on the grammar, the parser is guaranteed to recover from any syntactic input error and will output a correct structure tree. In terms of efficiency, user friendliness and maintainability, the generated parser contributes significantly to the quality of software containing it. These design goals have been met with only a very modest space cost over parsers that have no error recovery.

7. ACKNOWLEDGEMENTS

Professor William Waite challenged me with this project and contributed with many ideas and improvements.

This work was supported in part by the National Bureau of Standards 70NANB5H0506, the Army Research Office DAAL 03-86-K-0100, and the Office of Naval Research N00014-86-K-0204.

8. REFERENCES

[Corbett1985] Corbett, R. P., "Static Semantics and Compiler Error Recovery", UCB/Computer Science Dpt. 85/251, Berkeley, June 1985.

[Dencker1985] Dencker, P., *User Description of the Parser Generating System PGS*, Institut fur Informatik Universitat karlsruhe, 1985.

[Fischer1980] Fischer, C. N., D. R. Milton and S. B. Quiring, "Efficient LL(1) Error Correction and Recovery Using Only Insertions", *Acta Informatica 13* (1980), University of Wisconsin-Madison.

[Graham1982a] Graham, S. L., C. B. Haley and W. N. Joy, "Practical LR Error Recovery", *SIGPLAN Notices*, 1982.

[Graham1982b] Graham, S. L., P. B. Kessler and M. K. McKusick, ''gprof: a Call Graph Execution Profiler'', *UNIX PROGRAMMER'S MANUAL 4.2BSD, Vol 2C*, 1982.

[Gray1985] Gray, R. W., ''Comparing Semantic Analysis Efficiency of a GAG Generated Compiler vs Hand Written Compilers'', ECE690 Report, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO, Dec. 1985.

[Gray1987] Gray, R. W., *Generating Fast, Error Recovering Parsers*, University of Colorado, Computer Science Dept., Boulder, CO, Apr. 1987. M.S. Thesis.

[Gries1976] Gries, D., ''ERROR RECOVERY and CORRECTION - An Introduction to the Literature'', *Compiler Construction - An Advanced Course Edited 2nd Ed*, 1976.

[Hoare1981] Hoare, C. A. R., ''The Emperor's Old Clothes'', *Comm. of the ACM 24*, 2 (Feb. 1981).

[Horning1976] Horning, J. J., ''What the Compiler Should Tell the User'', *Compiler Construction - An Advanced Course Edited 2nd Ed*, 1976.

[Johnson1979] Johnson, S. C., ''YACC- Yet Another Compiler Compiler'', *UNIX PROGRAMMER'S MANUAL Seventh Edition, Vol 2B*, Jan 1979.

[Machanick1986] Machanick, P., ''Are LR Parsers Too Powerful?'', *SIGPLAN Notices 21*, 6 (June 1986).

[Pennello1986] Pennello, ''Very Fast LR Parsing'', *SIGPLAN Notices 21*, 7 (July 1986).

[Rohrich1980] Rohrich, J., ''Methods for the Automatic Construction of Error Correcting Parsers'', *Acta Informatica*, 1980.

[Waite1983] Waite, W. M. and G. Goos, *Compiler Construction*, Springer-Verlag, 1983.

[Wirth1985] Wirth, N., ''From Programming Language Design to Computer Construction'', *Comm. of the ACM 28*, 2 (Feb. 1985).

END
DATE
FILMED
DTIC
6-88